Acceleration of a Full-scale Industrial CFD Application with OP2

I.Z. Reguly, G.R. Mudalige, M.B. Giles, University of Oxford
C. Bertolli, A. Betts, P.H.J. Kelly
Imperial College London, (IBM TJ Watson)
David Radford
Rolls-Royce plc.
The Challenge

• HPC is undergoing an enormous change
  – New hardware architectures
  – New parallel programming abstractions, languages
• Flat (MPI) parallelism -> Multiple levels of parallel programming
• Getting high performance means specialization for the hardware
• Code maintainability, longevity
• “Future proofing”
Domain Specific Languages

• Separate abstract specification of computations from the parallel implementation
• High productivity for the domain scientist
• High productivity for the library developer
  – Can experiment and validate on small benchmarks, results immediately apply to large-scale scientific codes
• As hardware changes, the library adopts the latest and greatest features, optimizations
  – “User” code doesn’t change
Domain Specific Languages

• Lots of research done on DSLs
  – Most of them wither away and die...

• What are the obstacles to widespread adoption?
  – Critical mass
  – Usually applied to simple, toy problems
  – Little evidence that DSLs can be applied to industrial scale applications
Unstructured Meshes

- For extremely complex cases, unstructured meshes are the only tool capable of delivering correct results.
- Large, very complicated codebase

Vorticity isosurface from a large Eddy simulation of a compressor

Ground vortex ingestion
OP2 for Unstructured Grids

• Abstraction:
  – Sets, maps, data
  – Loop over sets, describing access type

res.h:
void res(double *A, double *u, double *du) {
    (*du) += (*A) * (*u);
}

... Call “res” for each edge Iterate over edges
op_par_loop(res,"res", edges,
    op_arg_dat(A,-1,OP_ID, 1,"double",OP_READ),
    op_arg_dat(u, 0,col,1,"double",OP_READ),
    op_arg_dat(du,0,row,1,"double",OP_INC));

With the following arguments
Rolls-Royce Hydra

Hydra is an unstructured mesh production CFD application used at Rolls-Royce for simulating turbo-machinery of aircraft engines.

- Turbines
- Noise
- Internal Engine blades
- Full aircraft
Rolls-Royce Hydra

- Used for the design of turbomachinery
  - Key CFD production code
  - Steady and unsteady flow
  - Reynolds Averaged Navier-Stokes
- In development for >15 years
  - Fortran 77
  - 50k+ lines of source code
  - ~300 computational loops
- Written in OPlus – same notions of sets, maps, data and loops over sets
- Our goal is to evaluate the utility of OP2, when applied to Rolls-Royce Hydra
Conversion

• The original source code had to be converted to use the OP2 API, keeping the “science” intact

• Hydra was based on OPlus, the conversion was not difficult
  – Computations did not change, they were only outlined and described using the parallel loop API
  – Constants were put in F90 modules
do while(op_par_loop(ncells, istart, iend))
  call op_access_r8(’r’, areac, 1, ncells, & null, 0, 0, 1, 1)
  call op_access_r8(’u’, arean, 1, nnodes, & ncell, 1, 1, 1, 3)
  do ic = istart, iend
    i1 = ncell(1,ic)
    i2 = ncell(2,ic)
    i3 = ncell(3,ic)
    arean(i1) = arean(i1) + areac(ic)/3.0
    arean(i2) = arean(i2) + areac(ic)/3.0
    arean(i3) = arean(i3) + areac(ic)/3.0
  end do
end while

subroutine distr(areac, arean1, arean2, arean3)
  real(8), intent(in) :: areac
  real(8), intent(inout) :: arean1, & arean2, arean3
  arean1 = arean1 + areac/3.0
  arean2 = arean2 + areac/3.0
  arean3 = arean3 + areac/3.0
end subroutine

op_par_loop(cells, distr, & op_arg_dat(areac, -1, OP_ID, 1, OP_READ), & op_arg_dat(arean, 1, ncell, 1, OP_INC), & op_arg_dat(arean, 2, ncell, 1, OP_INC), & op_arg_dat(arean, 3, ncell, 1, OP_INC))
Code generation

- OP2-Hydra can do pure MPI right away, but performance is poor due to loss of optimisations (function pointers, outlined code, going through Fortran to C bindings)
- Code generation for MPI can recover these optimisations
- Code generation necessary for OpenMP and CUDA execution
- Python script parses op_par_loop calls in high-level files, replaces them with calls to generated code
Initial code structure

Main application files

```fortran
! in file flux_app.F90
program flux_app
use OP2_Fortran_Reference
use OP2_CONSTANTS
use FLUX
...
call op_decl_set (nodes, ..)
call op_decl_map (..)
call op_decl_dat (x, ..)
...
call op_par_loop(nodes, flux_user_kernel,
  & op_arg_dat(x, -1, OP_ID, 3, OP_READ),
  & ...)
...
end program flux_app
```

Outlined computational kernel

```fortran
! in file flux.F90
module FLUX
subroutine flux_user_kernel(x, ...)
  real(8) x(3)
  ...
end subroutine
end module FLUX
```
Code structure after code generation

```fortran
! in file flux_app_op.F90
program flux_app_op
use OP2_FORTRAN_DECLARATIONS
use OP2_Fortran_RT_Support
use OP2_CONSTANTS
use FLUX_GENSEQ

... call flux_kernel (nodes,
   & op_arg_dat (x, -1, OP_ID, 3, OP_READ),
   & ...)
...
end program flux_app_op
```
Code structure after code generation

```fortran
module FLUX_GENSEQ

subroutine flux_user_kernel(x, ...)
real(8) x(3)
...
end subroutine

subroutine flux_wrap(arg1data,...,nelems)
real(8), arg1data(3,*)
do i = 0,nelems-1,1
   call flux_user_kernel(x(1,i+1),
   & ...)
enddo
end subroutine

subroutine flux_kernel(arg1, ...)
real(8), dimension(:,), pointer :: arg1Ptr
...
call c_f_pointer(arg1%CPtr, arg1Ptr, ...)
...
call flux_wrap(arg1Ptr,...)
...
end subroutine
end module FLUX_GENSEQ
```

- Computational kernel and loop over set in the same source file
- Workaround for Fortran pointer issues
- Fully automated, just python string manipulation
Baseline performance

OPlus PP vs. OP2 perfectly match, down to instruction count being within 5%.

[Graph showing performance comparison]

2 socket
Xeon E5-2640
2*12 cores
2.4GHz
Basic optimisations in OP2

- Support for ParMetis and PT-Scotch partitioning
- Partial halo exchanges for boundary loops
- Mesh renumbering to improve cache locality

![Bar chart showing performance comparison between OPlus, OP2, +PTScotch, and +renum for different numbers of threads.](chart.png)

- 22 sec
- 17 sec

2 socket
Xeon E5-2640
2*12 cores
2.4GHz
Heterogeneous execution

- Support for MPI+OpenMP and MPI+CUDA
- Code generation + pre-processing to support shared memory parallelism via colouring
  - Partitions further broken up into mini-partitions/blocks, each coloured. Blocks of the same colour can be executed concurrently
  - Colouring within blocks enables GPU execution where different threads process different elements of a block
Further optimisations

- Block size determines cache locality, work granularity and load balancing
- With OpenMP NUMA effects, with the GPU occupancy
- Auto-tuning is essential!

![Bar chart showing performance times for different thread configurations.](chart.png)

- 2 socket Xeon E5-2640
- 2*12 cores
- 2.4GHz
GPU optimisations

- Some automatically performed during code generation, some need manual tweaking
  - Block sizes
  - Reading through read-only cache by adding `intent(in)`
  - Transposed data layout, either fully automatic, or by adding "`:soa" to type descriptions in `op_decl_dat` and `op_arg_dat`

![Bar chart](chart.png)

- OPlus (24 MPI)
- 24 MPI
- CUDA (initial)
- CUDA (+ SoA)
- CUDA (+ block opt)
- 1 MPI x CUDA (best)
- 2 MPI x CUDA (best)

2 socket
Xeon E5-2640
2*12 cores
2.4GHz
+ Tesla K20c

Tesla K40:
8.8 sec
Strong scaling

800K vertices, 2.5M edges. 1 Hector node (32 cores) and 1 Jade node (2 K20 GPUs)

Linear scaling up to 16 nodes (512 cores)
Strong scaling

800K vertices, 2.5M edges. 1 Hector node (32 cores) and 1 Jade node (2 K20 GPUs)

Runtime (Seconds)

Nodes

OPlus

OP2 MPI (RCB)

OP2 MPI (PTScotch)

Linear scaling up to 16 nodes (512 cores)
Strong scaling

800K vertices, 2.5M edges. 1 Hector node (32 cores) and 1 Jade node (2 K20 GPUs)

Linear scaling up to 16 nodes (512 cores)
Strong scaling

800K vertices, 2.5M edges. 1 Hector node (32 cores) and 1 Jade node (2 K20 GPUs)

Runtime (Seconds)

Nodes

Linear scaling up to 16 nodes (512 cores)
Strong scaling

Nodes

Run time (milliseconds)

vfluxedge
ifluxedge
edgecon
accumedges
srcsa
updatek
period
d periodicit
Weak scaling

0.5M vertices per node

GPU node has 2* over HECToR node
Weak scaling

0.5M vertices per node

GPU node has 2* over HECToR node
Weak scaling

0.5M vertices per node

![Graph showing runtime in seconds for different configurations across various nodes.]

- OPlus
- OP2 MPI (PTScotch)
- OP2 MPI (RCB)
- OP2 MPI+OMP (PTScotch)
- OP2 MPI+OMP (RCB)

GPU node has 2* over HECToR node
Weak scaling

0.5M vertices per node

GPU node has 2* over HECToR node
Weak scaling

Run time (milliseconds)

Nodes

accumedges
ifluxedge
updatek
period
edgecon
srcsa
vfluxedge
periodicity
Hybrid CPU-GPU execution

• Using the CPU and the GPU at the same time
• Some processes use the CPU, some the GPU
• How to load balance? Some loops are faster on the GPU, some on the CPU
Conclusions

• DSLs can be applied to industrial-scale codes
• Early version was slow: cost of a high-level API
  – Had to understand these limitations, code generate to circumvent them
• Matching & increased performance on the same HW
  – By using OP2, some improved techniques come for “free” (renumbering, better partitioning, better MPI, etc.)
• Enabled OpenMP, CUDA and CPU+GPU Hybrid execution
  – On such complicated code, the performance advantage is not huge – but the option is there!
• All of these optimisations apply with no (or very little) change to the user code
Thank you!

Questions?

istvan.reguly@oerc.ox.ac.uk

Acknowledgements:
This research has been funded by the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project, the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on “Multi-layered Abstractions for PDEs” and the “Algorithms, Software for Emerging Architectures“ (ASEArch) EP/J010553/1 project. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work.

Special thanks to: Brent Leback (PGI), Maxim Milakov (NVIDIA), Leigh Lapworth, Paolo Adami, Yoon Ho (Rolls-Royce), Endre László (Oxford), Graham Markall, Fabio Luporini, David Ham, Florian Rathgeber (Imperial College), Lawrence Mitchell (Edinburgh)
GPU optimisations

• Through the code generator
  – Replace device constants (regexp)
  – Change to SoA access (regexp)

\[ \text{var}(m) \rightarrow \text{var}(\text{nodes\_stride} \times (m-1)+1), \text{through Op2\_SOA}(\text{var, nodes\_stride,m}) \]

• Manually
  – Add intent(in) to variables to enable caching loads

• Auto-tuning
  – Block sizes, register counts